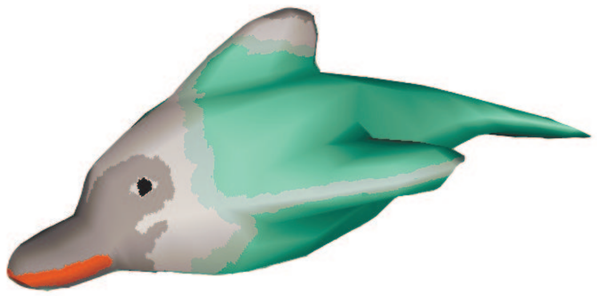
A 3D Interactive Texture Painter

George ElKoura
University of Toronto
gelkoura@dgp.toronto.edu

Abstract

This paper presents a method for painting texture interactively on a 3D model. Unlike some previously published methods, the technique presented here does not require a grid topology and does not require a dense mesh. The resolution of the texture map produced is independent of the density of the geometry.
We also discuss the application in a 3D application of various brushes traditional to 2D painting. We also present ideas that are only possible in 3D.



1. Introduction

It is common in computer graphics to take shortcuts in order to generate a believable image efficiently. Texture maps are the most commonly used among such shortcuts. They allow us to modulate the properties for any point on a surface. Texture maps are simply a bitmap that can be applied to a surface. The bitmap can contain diffuse colour values, alpha (transparency) values, lighting and shadow information (usually called light and shadow maps respectively in that context). Texture maps help us to efficiently create realistic looking computer imagery.

The difficulty in using texture maps is that for many surfaces used in computer graphics, there is no natural parameterization that helps us predictably map a 2D bitmap on a 3D model. Most notably, a general polygonal model has no obvious parameterization whatsoever.

The problem of parameterization is currently solved by painstakingly mapping each vertex of the polygonal mesh to a point on the 2D texture. Many commercial software products are available to help with this. Usually, a spherical or cylindrical projection is used on the mesh (depending on the shape of the object) and then the texture map is tweaked manually to correct the errors of the simple projection.

Needless to say this is a tedious process. Some studios hire artists dedicated to this task alone.

This paper presents a method for painting the texture map directly on the 3D surface interactively. The user never has to worry about parameterization or texture-map distortion because what they see on the screen is the final result of the texture mapping.

Section 2 discusses previous work in this area. Section 3 discusses the topology required by the program and the corresponding texture map that is generated. Sections 4 and 5 discuss some ideas for brushes that can be used to paint the texture. Section 6 discusses hardware acceleration techniques that make this application interactive. Finally, section 7 discusses results and section 8 provides a conclusion and a discussion of future work.

2. Previous Work

Pat Hanrahan and Paul Haeberli described a program [1] that can alleviate some of the tedium in texture mapping by allowing the artist to paint directly on the object.

Their system allows the artists to paint material attributes interactively on the model. The system is not confined to any particular material property and can be used to paint diffuse colour as readily as to paint specular colour or transparency and even geometric displacement.

Their system is dependent on a geometric topology that can be mapped to a regular grid. This gives a simple parameterization with which to work and removes some complexity at the expense of generality.

Another program that uses direct 3D texture mapping is produced by Right Hemisphere called Deep Paint 3D. However, we couldn't find any published papers describing their work.

3. Texture and Topology

Two main extensions to Hanrahan and Haeberli's paper were the drive behind the program presented in this paper. The first was to remove the grid topology restriction and the other was to remove the dependency between mesh density and texture resolution.

Our program works on any topology, even non-manifold topologies. The only restriction imposed on the model is that it be composed of quadrilateral polygons. Although, we could have easily allowed for triangles as well, we chose to only support quadrilaterals because objects represented as Catmull-Clark subdivision surfaces [4] are in such a form and are easy to model.

The density of the mesh and the resolution of the texture are independent in our program. This is important because modern hardware can more easily render a low-density mesh with high-resolution texture than a high-density mesh. This is accomplished by generating an actual bitmap for the texture.

A first attempt at solving this problem involved allocating a separate texture map for each polygon in the mesh. Needless to say, this quickly proved to be inefficient and rendered the program too slow for interactive application on objects that contain even a modest number of polygons.

The solution currently implemented, is to allocate a single texture map (of high-resolution) and assign a region of the texture map to each polygon. The parameterization of the polygons and this region allocation is done at the same time. Each polygon gets texture coordinates that correspond to its region in the large texture map. Note that in this method, polygons that are adjacent in object space do not necessarily get adjacent regions in the texture map. This is what removes the topological restrictions imposed by earlier work.

It is important to note here that due to precision errors in assigning the texture coordinates, it may be more effective to divide the geometry into a small number of texture maps rather than using a single texture map.

The advantage in [1] of closely tying the texture resolution to the mesh density is that it can very simply implement geometric displacement in real-time. Our system does not allow for this because of current hardware limitations. Once displacement shaders can be implemented in hardware, then our system will also be able to do this more efficiently. Currently, our system does not preclude painting bump maps or displacement maps that can be used in an external renderer that supports these features.

3.1 Gutter Space

In an initial implementation, the problem of "bleeding" was very bothersome. Bleeding occurs when the texture regions are laid out too close to one another. The problem is particularly annoying because texture region adjacency does not correspond to geometric adjacency, so bleeding could occur on a polygon far away and disconnected from where we are painting. To solve this problem, we create gutter space to separate the texture region. Since the assignment of texture coordinates occurs independently of the texture map allocation, the gutter space is currently assigned as a small fraction.

4. 2D Brushes Applied to 3D

2D paint programs are well established, and many of the techniques used in 2D programs have application in a 3D paint program. However, some complications arise in 3D that aren't present in 2D. The main issue is that we have two spaces: screen space and object space. Our implementation currently has four brushes that are borrowed from 2D paint programs and applied to 3D.

4.1 Local Space Brush

The local space brush operates only on the local space of the geometry. Once the polygon under the cursor is found, and we know what part of the texture map we'd like to modify (see Section 6 for how this is done), this brush paints a filled circle using the current colour and clips the circle to the region of the texture map occupied by the polygon. Of importance to note here is that the position under the cursor is sampled only once.

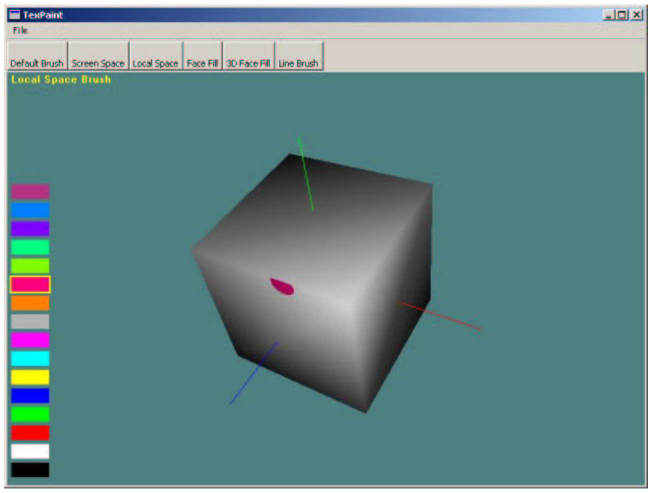


Figure 4.1.1 Local Space Brush. *Notice how the brush nib is clipped at the cube face boundary because the mouse was clicked inside the face. Compare this with the default brush.*

4.2 Screen Space Brush

The screen space brush operates by effectively drawing a circle in screen space and projecting down onto the mesh. Here a midpoint algorithm for drawing a circle is used to generate a list of pixels in screen space. Then each pixel is sampled to get the polygon and the texture coordinates at that pixel and then that pixel is painted. Due to the low precision of floating points used on the hardware, this does not produce desirable results. Many pixels map to the same texture coordinates because of the low-precision interpolation. This produces a speckled look, which is usually not what the user expects. See section 6 for a discussion on the effect and possible solution to the lack of desired precision on the hardware.

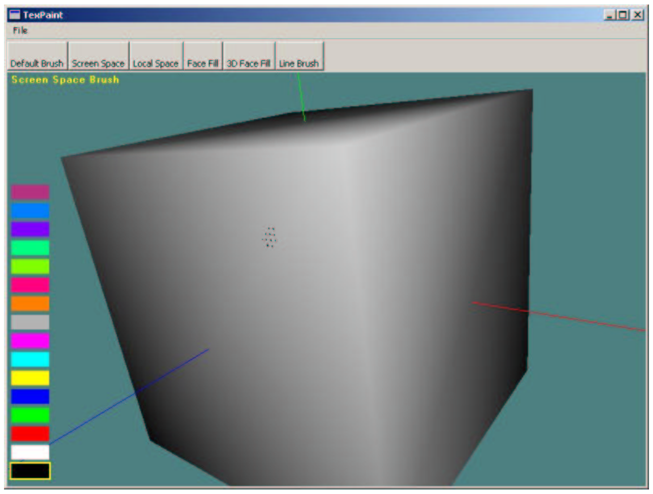


Figure 4.2.1 Screen Space Brush. *Here we can clearly see the problem. The hardware interpolates using only 8 bits per channel, thus giving us only a few points on the texture map that we can colour.*

4.3 Default Brush

For lack of a better name, the default brush operates both in screen space and in local space. When it finds the pixels we need to paint in screen space, we then paint a circle in local space. However, since the circles we draw in local space are still dependent on the sampled pixels and texture coordinates, this brush still suffers from the lack of precision on the graphics hardware, but to a much lesser degree than the screen space brush.

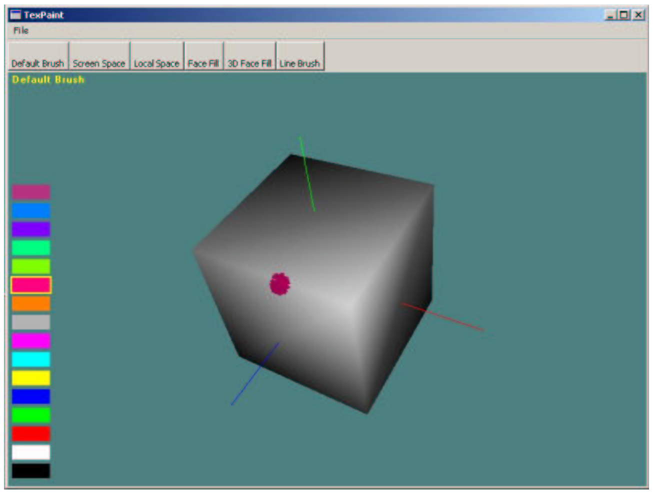


Figure 4.3.1 Default Brush. *Notice how the brush nib is not clipped at the cube face boundary but looks like a circle projected onto the geometry. Compare this with the local space brush.*

4.4 Line Brush

The line brush implements a simple Bresenham line algorithm in screen space and applies the same technique in the default brush along the rasterized line. This allows us to draw straight lines on screen and have the line projected onto our model.

This same technique can be applied for virtually all simple 2D brushes, for example, circle, ellipse, rectangle, and so on. Only the line brush is currently implemented.

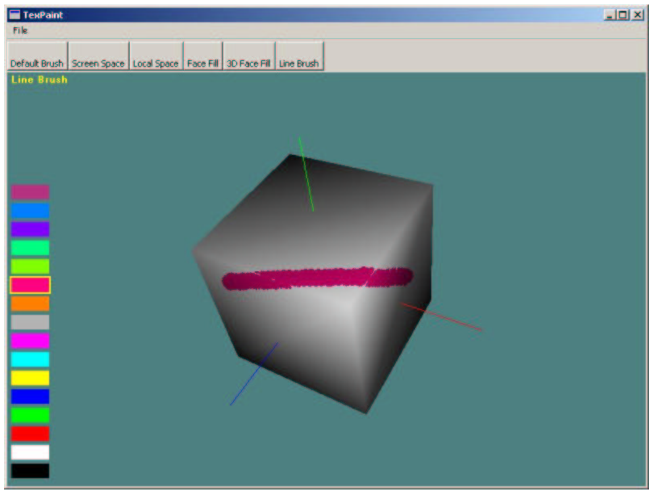


Figure 4.4.1 Line Brush. *This is the line brush, which uses an underlying default brush. Here the line is projected onto the geometry*.

5. 3D Brushes

In three dimensions we can offer interesting brushes that make our job of painting a 3D surface simpler. We have implemented two such brushes, where one is simply the generalization of the other.

5.1 Face Fill

The face fill brush simply colours the polygon under the pixel uniformly with the selected colour. The 3D face fill brush provides a generalization of this brush.

5.2 3D Face Fill

The 3D face fill brush behaves like the simple face fill brush in that it colours a face in a uniform selected colour. However, it also allows the user to select an angle and it continues to colour polygons whose normals differ by less than the selected angle. This allows the user the uniformly colour a region that contains a little curvature. Co-planar polygons can be coloured in this way by setting the angle to 0, or the entire mesh can be coloured by setting the angle to 90 (given that your mesh meets the restriction.)

For this brush, it was necessary to implement a structure for connectivity so that face fills don't jump over discontinuities. We implemented a simple point connectivity structure. We believe that more predictable results would be obtained from an edge connectivity structure such as the Baumgart's winged edge structure [5].

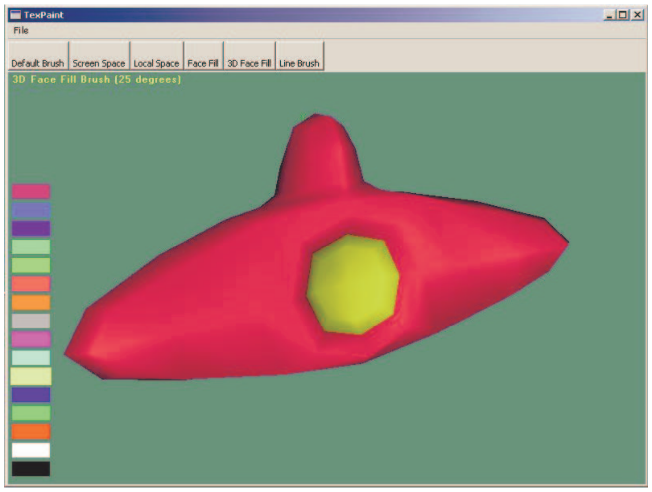


Figure 5.2.1 3D Face Fill Brush. *This is the geometry of a toy spaceship. Using the 3D Face Fill brush with the angle parameter set to 90 degrees, the entire object was painted red, then using an angle of 25 degrees, only the burner area was painting in yellow, because the polygons comprising this area don't defer by more than 25 degrees.*

6. Hardware Acceleration

One of the main problems that must be solved is performing the inverse mapping from the position on the screen where the user clicked to the texture coordinates of the polygon that occupies the pixel.

Our first attempt used the object tag technique described in [1] where the entire geometry was rendered into a buffer, but instead of colour using an integer that uniquely identifies the polygon. On modern hardware we have at least 24 bits to represent the unique ID and can thus support a good number of polygons. Once the polygon is discovered we then cast a ray from the eye through the pixel and intersect it against the polygon. Only the selected polygon is intersected so we do not need to perform any ray tracing optimizations. Once the point on the ray that intersects the polygon is found, we need to convert this point into parameterized space of the polygon. This is the inverse parameterization required to find the texture coordinates corresponding to the point. Since we only support "well-behaved" quadrilaterals, we triangulate the quad and find the barycentric coordinates of the point in whichever triangle contains it.

This method works, but it is slow. We can optimize the entire process by taking advantage of modern hardware and our imposed texture coordinates. Our texture coordinates are designed in such a way that no two polygons have any texture coordinates in common. In other words, the texture coordinates uniquely identify the polygon. Thus, instead of rendering a unique ID during the picking phase, we instead use the red and green channels to render the u and v of the texture coordinates respectively. Thus we let the hardware interpolate the texture coordinates for us. All we have to do now is read this buffer to get the texture coordinates corresponding to where the mouse cursor was positioned.

A very important note here is that the precision is still 24 bits, and since we are only using red and green we are limited to 16 bits of precision. This is not adequate for practical purposes and is the cause of the problem in the screen space brush discussed earlier. We get around this problem by painting an area instead of just a point. However, the ideal solution would come from true floating-point operations on the graphics hardware. The next generation graphics hardware promises to offer such support.

7. Results

The techniques presented here work well at interactive speeds on faster machines. Older-generation machines with poor texture hardware seem to struggle during the painting. We could perhaps implement methods where we decrease the texture resolution while the user is painting if we require that the program run on the older machines.

The application is fun to use but without more brushes, it is hard to produce production quality textures. As with other applications of this nature, it requires a skilled artist to produce visually appealing results.

The main dissatisfaction is with the low-precision hardware interpolation used to perform the texture coordinate calculations. This limitation makes the default brush hard to predict and hard to use for detailed work. Being able to control the brush sizes helps quite a bit, but still does not alleviate the need for higher precision calculations on the hardware.

8. Conclusions and Future Work

The techniques discussed in this paper are a stepping-stone towards what could be a very useful and entertaining application for painting interactively on the surfaces of 3D objects. Removing the need for the artists to manually assign texture coordinates, lifting the grid topology restriction, and separating texture density from mesh density all go towards making texture mapping accessible even to young children.

See Figures 8.1 and 8.2 for some images of the program at work.

The important next steps are implementing many more brushes and improving the texture coordinate interpolation precision, either through hardware or through a fast software solution.

Also, more intelligent allocation of texture coordinates would be helpful. For example, a polygon with bigger area would occupy a larger texture map region. This would also help in dynamically selecting the texture map resolution, which is now done statically.

Independent to the dynamic texture map resolution is being able to calculate and use the optimal size of the gutter for the texture map.

9. References

[1] P. Hanrahan and P. Haeberli, "Direct WYSIWYG Painting and Texturing on 3D Shapes", *In ACM SIGGRAPH Conference Proceedings*, August 1990, pp. 215-223.

[2] D. Hearn and M. P. Baker, "Computer Graphics", 2nd Ed, Prentice Hall, 1997.

[3] M. Woo, J. Neider et al., "OpenGL Programming Guide", 3rd Ed, Addison-Wesley, 1999.

[4] E. Catmull and J. Clark, "Recursively Generated B-Split Surfaces On Arbitrary Topological Meshes", *Computer Aided Design*, 1978, pp. 350-355.

[5] B. Baumgart, "Winged Edge Polyhedron Representation", Technical Report CS-320, *Stanford Articial Intelligence Laboratory*, 1972.

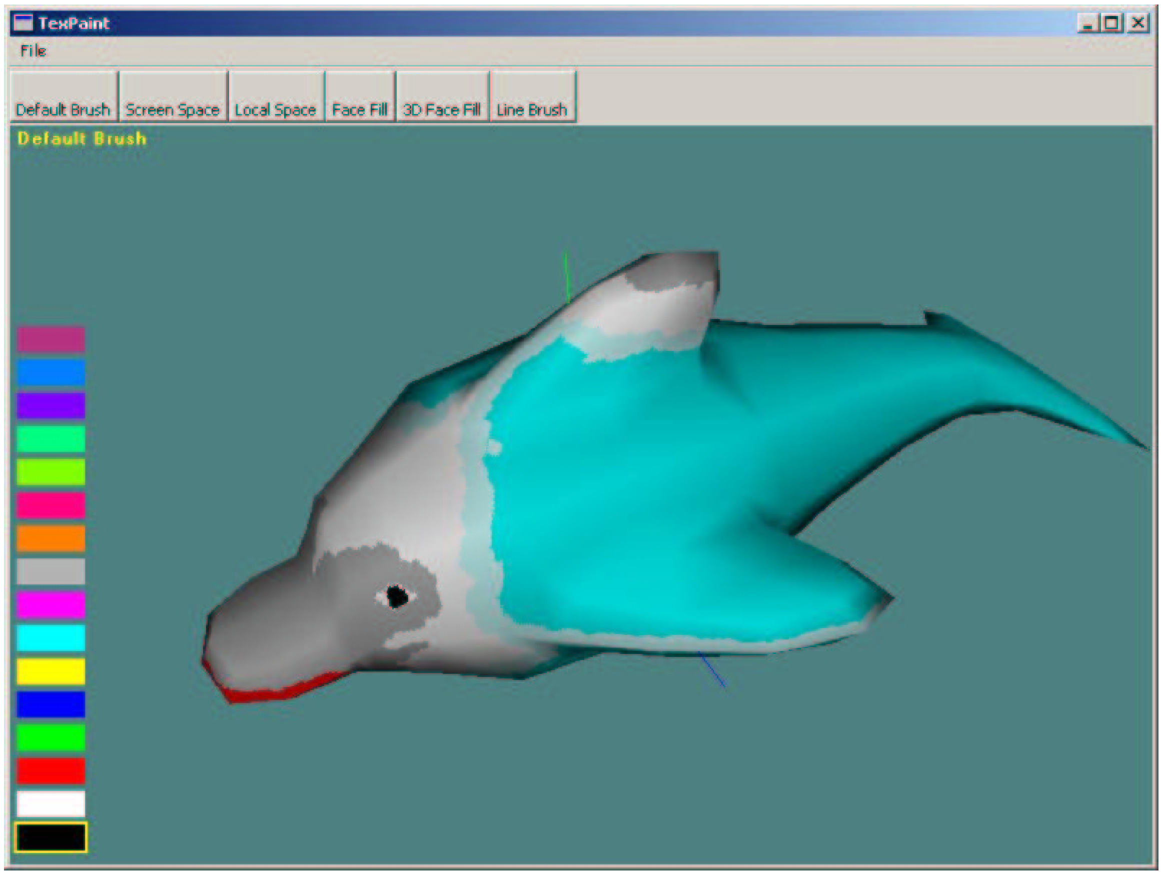


Figure 8.1 Dolphin Screenshot. *This image shows the application painting a texture on a model of a dolphin modeled using subdivision surfaces. See Figure 8.2 for the texture map that is generated for this model.*

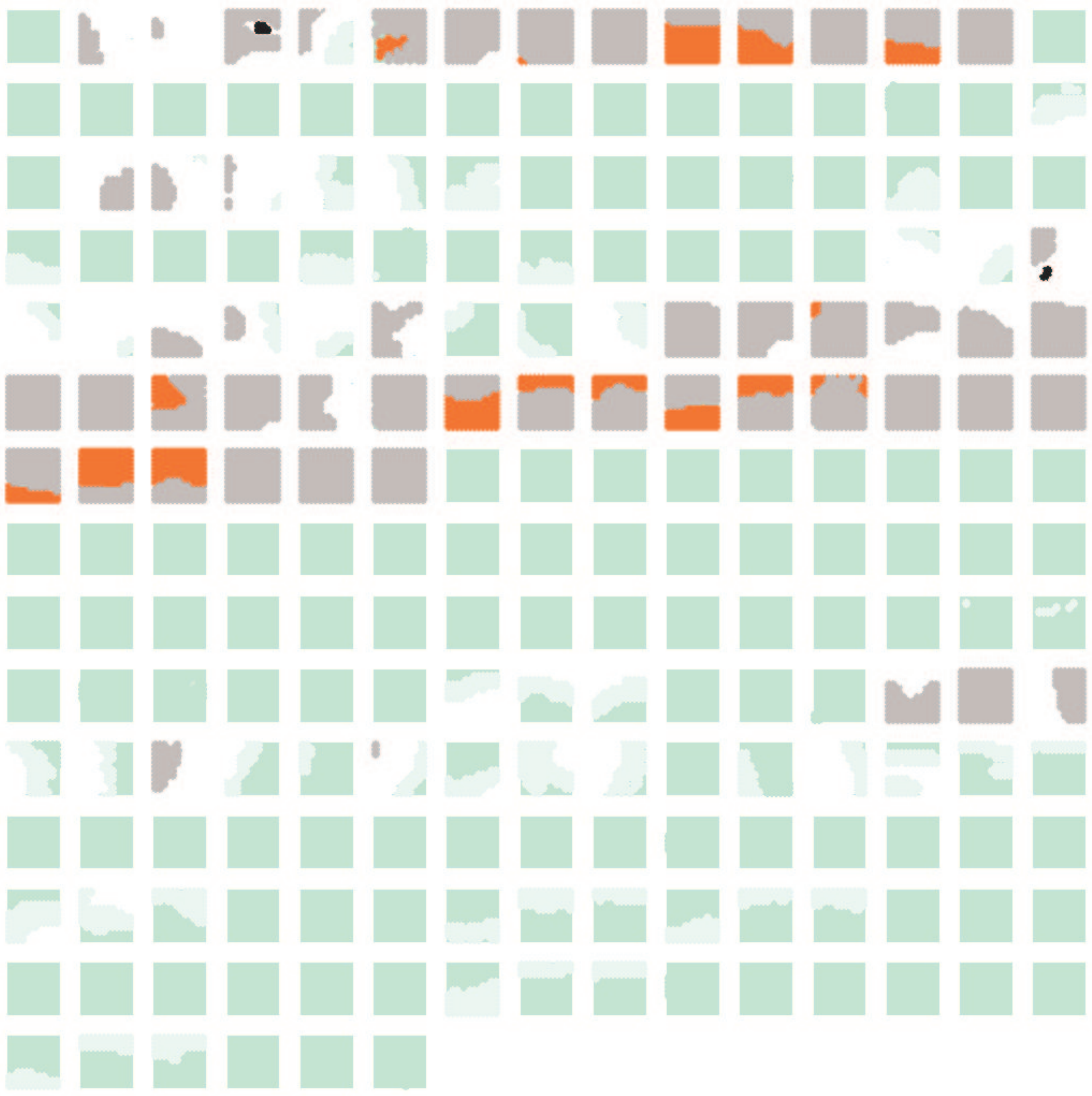


Figure 8.2 Dolphin Texture Map. *This is the texture map that is generated for the dolphin model shown in Figure 8.1. Note that the "gutter" space between the texture regions is exaggerated here for clarity.*